Universität Augsburg


# Model-Driven Development of Secure Service Applications introduced by a Banking System Example

**Marian Borek, Kurt Stenzel, Nina Moebius, Wolfgang Reif**

Report 2012-03 March 2012


Institut für Informatik
D-86135 Augsburg

**Abstract**

SecureMDD is a model-driven approach to develop security-critical systems. It uses a model that represents an application and generates executable code as well as a formal specification that can be used to prove security properties. In our previous works we considered secure smart card application and now we focus on secure service applications. In this report we describe the modelling of services, their communication and their security. We also show the generated code and explain the integration of services in the formal model. To demonstrate our approach we show a banking system which uses smart cards and services.

# 1 Introduction

The worldwide enterprise software revenue in 2011 is estimated to $267 Billion [8], but only 32% of all software projects were successful [9]. The reasons for the failure of a software project vary, but often lack of time and complexity are important issues. Model-driven software development that uses a model to create a software system can make the development process faster and easier. For that, a model will be created that enables a high abstraction of the system. The implementation can be generated automatically or semi-automatically from this model. Furthermore, during the build, documents that record the states of the development will be created. Especially the development of security-critical systems is very difficult and error-prone. With the model-driven approach the model of the system is available and with correct transformations it is ensured that the model is equivalent to the generated code. Based on this fact it is possible to verify properties for the model that are also valid for the implementation. Therefore developing security-critical software with a model-driven approach has large potential.

Our project SecureMDD is an approach for developing security-critical systems with UML. From a model that represents an application, executable code as well as a formal specification which is used to prove security properties is generated. Especially applications that store or transmit sensitive data like money require verification to prove their security. We focus on applications which interact with humans and handle security critical data. Some of our case studies are business applications but we are not restricted to this area of application.

Until now we considered only security critical smart card applications. In this report we present how security critical service applications have to be handled in a model driven approach. Services differ a lot from smart cards because of the different communication abilities, the different behaviour and the different operational areas. For developing security critical service applications in a model driven approach it has to be clarified how services, their communication and their security are integrated in the model, generated code and formal specification. In detail, it has to be determined how to handle concurrent service invokes and how to support sessions for state based protocols. Also, it has to be worked out how to secure transferred messages between any possible communication participants, store complex data and how to generate executable service code.

This report is structured as follows. Section 2 gives an overview of our SecureMDD approach and section 3 describes the modelling of services. Section 4 introduces the application example Debitcard which illustrates a bank system

application where the customers of different banks own a card to withdraw money from an ATM or transfer money from their accounts to others. Section 5 explains the generated code as well as the deployment. Section 6 explains the effects of services and their secure communication on the formal specification and gives an overview of proved application-specific security properties. Section 7 discusses related work and Section 8 concludes this paper.

# 2 The SecureMDD Approach

SecureMDD is a model-driven approach to develop security critical systems. Based on a model that represents a system, runnable code and a formal specification can be generated. The formal specification is used to verify application-specific security properties which the system has to satisfy. For example it could be required that a system or a part of it does not lose money, that security critical data will remain secret or that a Dolev-Yao [7] attacker cannot harm the system.

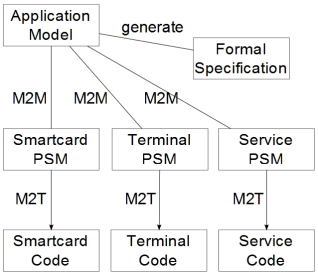


Figure 1: SecureMDD Approach

The SecureMDD approach (see Figure 1) uses a platform independent UML model and a UML profile as well as a platform independent and easy to use modelling language MEL [18] to define security-critical applications. For representing the system we use class diagrams, activity diagrams, sequence diagrams and a deployment diagram.

The class diagrams are used to describe the static part of the system and define classes with attributes and methods which can be used with stereotypes from our UML profile. These classes represent transferred messages or components like smart cards, terminals and services. The activity diagrams are used for the dynamic behavior and describe the cryptographic protocols with components and their messages as well as their behavior after receiving a message. The sequence diagrams represent a higher abstraction of the dynamic behavior and the deployment diagram is used to describe the communication structure, attacker abilities and connection security.

Based on the platform independent application model, a formal specification and three platform specific models – one for each component type – will be generated. The formal specification is the basis for the verification of security properties that can be proved with the theorem prover KIV [1]. The platform specific models are tailored to their target platforms, Java Card for a smart card component, Java for a terminal, and Java based Web services for a service component.

We define a smart card as an component that can be accessed only via a predefined interface and is tamper-proof: nobody has access to the operating system or the internal memory directly. A terminal can be a secure automaton, a personal computer, or a mobile device that receives instructions from a user and can have an interface for the communication with a smart card. A service is a component that can be connected with terminals or other services over a network. Each service describes an interface and an application that runs on a server which can be only accessed via its interfaces.

For further processing of the application model we use the development environment Eclipse[1]. The formal specification is generated with the framework oAW[2] and the three platform specific models are created automatically using model-to-model (M2M) transformations with QVT [21]. Model-to-text (M2T) transformations based on oAW are used to generate the executable code. Each PSM is transformed to one or more Java packages that contains the full source code for each component type. The language for terminals and services is Java, while smart card code uses Java Card [24, 4], a version of Java tailored for smart cards with their severe resource limitations.

More details about our approach and applications modelled with SecureMDD can be found in [18, 16, 17].

# 3 Modelling of Services

A service is identified by the stereotype ≪Service≫ and can be stateful or stateless which is represented by the same-named stereotype properties (stateless is default). Stateless means that all invokers communicate with the same service instance. Stateful means that someone who invokes a stateful service for the first time opens a session and gets a new instance, and after the work is done the service can close the session. In this session the invoker communicates with the new service instance, which can store session dependent information like session keys. But for protocols which do not have to store session dependent information, we provide stateless services. Those stereotypes can be applied to classes in a class diagram and nodes in a deployment diagram. A class describes the static part of a component. With nodes and the intermediate connections the communication structure is defined. For every component one class and one node with the same name and the same stereotypes have to exist.

Activity diagrams describe the behaviour of the components. They use swimlanes to define the behaviour for a certain component. Every swimlane is associated with a class that describes a component. One activity diagram defines one protocol for component instances that exchange messages with other component instances. To communicate with a stateful service, at the beginning a session has to be opened and at the end the session should be closed. For that the stereotypes ≪openSession≫ and ≪closeSession≫ have to be applied to the UML actions send signal action respectively receive event action. When ≪openSession≫ is applied to a send signal action, a new session will be opened before the message is sent and after receiving a message with a receive event action, with the applied ≪closeSession≫ stereotype the session will be closed. Because it is ambiguous where a session should start and where it should end it is necessary to model that information.

To model communication of service applications we use activity- and deployment diagrams. The orchestration of a service is described through the swimlanes which represent this service. Choreography is not modelled separately. It is described with all messages that are represented with arrows between two swimlanes. But each protocol is modelled as one activity diagram and so all messages for each protocol are clearly represented.

[1] http://www.eclipse.org/

[2] openArchitectureWare: http://www.openarchitectureware.org/

A service can be invoked by terminals and services and is able to invoke other services. That make it possible to model complex service hierarchies and realize a service oriented system. But a service is also able to communicate with a smart card over an intermediate terminal.

For securing the communication between an invoker and a service we support Transport Layer Security (TLS). TLS will be modelled as the stereotype ≪TLS≫ that is applied to a connection in the deployment diagram and it supports the properties *MutualAuthentication* and *ServerSideAuthentication*. The implementation uses a Java library so we do not verify this protocol but use assumptions like confidentiality and integrity during the verification of the modelled system (see Section 6). Besides TLS we also support special security data types for encrypted, signed or hashed message parts as well as for keys and other data types. Those security data types are not part of a standard because Web Services and Java Card do not support a common standard.

# 4 Debitcard: A Bank System Application

Debitcard is a system where a card holder who knows the PIN of his card can withdraw money from an ATM or use a PC with internet connection to transfer money from one account to another. The application consists of two kinds of banks: an affiliated bank with multiple ATMs and a direct bank that can only be reached via a network, e.g. the internet or an ATM of an affiliated bank.

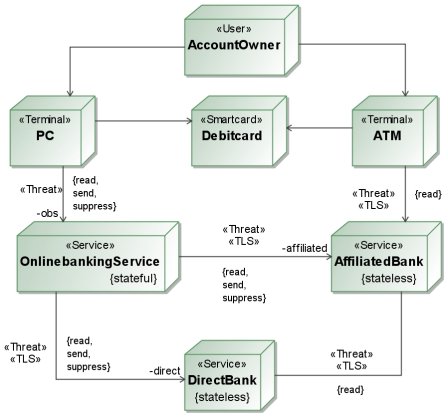


Figure 2: Communication structure of the Debitcard application

Figure 2 shows the deployment diagram that describes the communication structure, attacker abilities and connection security. There are four component types:

Terminal, Smartcard, Service and User, whereby the User represents a real person. Furthermore an attacker represented by the *Threat* stereotype can be applied to any connection. He can be a full Dolev-Yao attacker who is able to read, send, and suppress messages on the fly, but he can also have only a subset of these abilities.

The communication structure is defined with components and intermediate connections which can be unidirectional or bidirectional. The connection between *User* and *ATM* as well as between *ATM* and *Debitcard* is unidirectional. This means that every action is triggered by the User; only an *ATM* that has received an instruction from the *User* will send a message to *Debitcard*; afterwards the card can answer, but it cannot send messages of its own accord. In contrast, the connection between *DirectBank* and *AffiliatedBank* is bidirectional; this means that any of them can start a communication with the other if it has received a message from an *ATM* or an *OnlinebankingService*.

The attacker's abilities and connection security influence each other. If a connection has no applied *Threat* stereotype like the connection between *User* and *ATM* or *ATM* and *Debitcard* then the connection is assumed to be secure

(which must be achieved by physical means). Otherwise, if a connection has an applied ≪Threat≫ stereotype with any ability, the connection can be secured by security properties like ≪TLS≫ with the default property *MutualAuthentication*. As an example we consider the connection between *OnlinebankingService* and *AffiliatedBank* that has a ≪Threat≫ stereotype with the properties *read, send, suppress*. That means that the attacker can read, send and suppress messages on this connection. In combination with ≪TLS≫ it means that the attacker can only read encrypted messages, and that the properties send as well as suppress imply that the attacker can disconnect the connection (see Section 6). The assumption for the connection between *DirectBank* and *AffiliatedBank* is that the attacker can only read messages. Thus the attacker is not able to affect the connection or the transmitted messages (see Section 6).

Now we focus on services that can be stateful or stateless. *OnlinebankingService* is stateful. It has session dependent information like a protocol state and session key for every invoker. This way we are able to secure messages between *Debitcard* and *OnlinebankingService* with an application specific cryptographic protocol that uses *PC* only as an intermediate. The other services (*AffiliatedBank* and *DirectBank*) do not need to store session dependent information and thus it is sufficient that they are stateless.

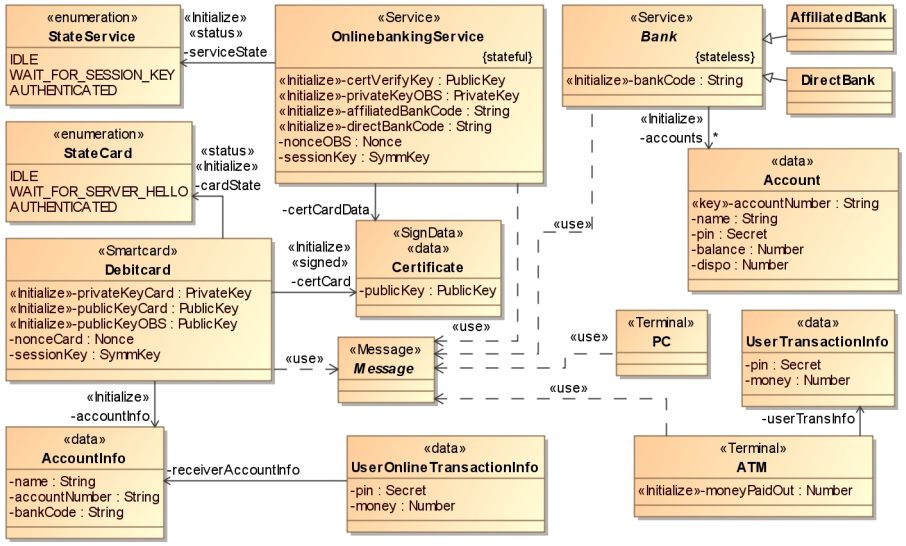


Figure 3: Class diagram of Debitcard

In Figure 3 a class diagram of Debitcard is illustrated. The classes describe the static part of the components from the deployment diagram (see Figure 2) and their attributes. There are some stereotypes that are applied to classes or attributes.

1. *Service, Terminal, Smartcard*: describe the component type
2. *Initialize*: means that this attribute has to be set during the deployment
3. *status*: can be applied to enumeration attributes and represent component states in a protocol

4. *sign*: can be applied on attributes with the type *SignData* and describe that this attribute is signed

5. *key*: is only legal in combination with a star association and causes that all elements in this set have a unique key element and can be identified through this one

There are also some predefined security data types that are necessary for developing security protocols in secure applications.

1. *PublicKey and PrivateKey*: public and private key for asymmetric encryption or signing data

2. *SymmKey*: a key for symmetric encryption

3. *Secret*: information that the attacker should never know

4. *Nonce*: an arbitrary number used only once to sign a cryptographic communication

The *OnlinebankingService* and the Debitcard employ cryptographic keys, states, nonces and a certificate to establish a secure communication between each other. Additionally, the *Debitcard* contains account information of an account owner who should be the owner of the card. The two components *AffiliatedBank* and *DirectBank* have an initial bank code and a list of accounts. They have no keys, states or nonces because their communication can be secured with an externally applied TLS protocol. The terminal *PC* has no attributes because it only forwards the messages between *OnlinebankingService* and *Debitcard*. An *ATM* needs to temporarily store the PIN and the money that an account owner try to withdraw. But it also needs to store how much money it has ever paid out. This information is important for the formal verification of the security property 'the money in the bank system remains constant'. All components need messages to interact with each other. For that each component has an association to the abstract message class *Message* form which all messages are derived.

The dynamic part of the system is described with activity diagrams. For sending and receiving messages we use send signal actions and accept event actions. To represent exceptions we use final flows and to describe actions like method calls or assignments we use actions that can also be used in structured activity nodes. For calling methods that are modelled in a subdiagram we use call behaviour actions. All described elements are standard elements of activity diagrams in UML 2.0. In guards, send signal actions, call behaviour actions and actions we use our abstract and platform independent Model Extension Language to describe behaviour.

The activity diagram for withdrawal of money from an ATM is pictured in Figure 4 and the message classes are shown in Figure 10 and Figure 9. First an *AccountOwner* who represents the card holder has to insert his card into an *ATM* slot and enter his PIN as well as the sum of money on the user interface of the *ATM* (1). The *ATM* stores this information in its attribute *userTransInfo*, asks the *Debitcard* for the account information (2) and sends all data to the *AffiliatedBank* (3). The bank first checks if the account belongs to itself or another bank by comparing the received bank code with its own one. If the

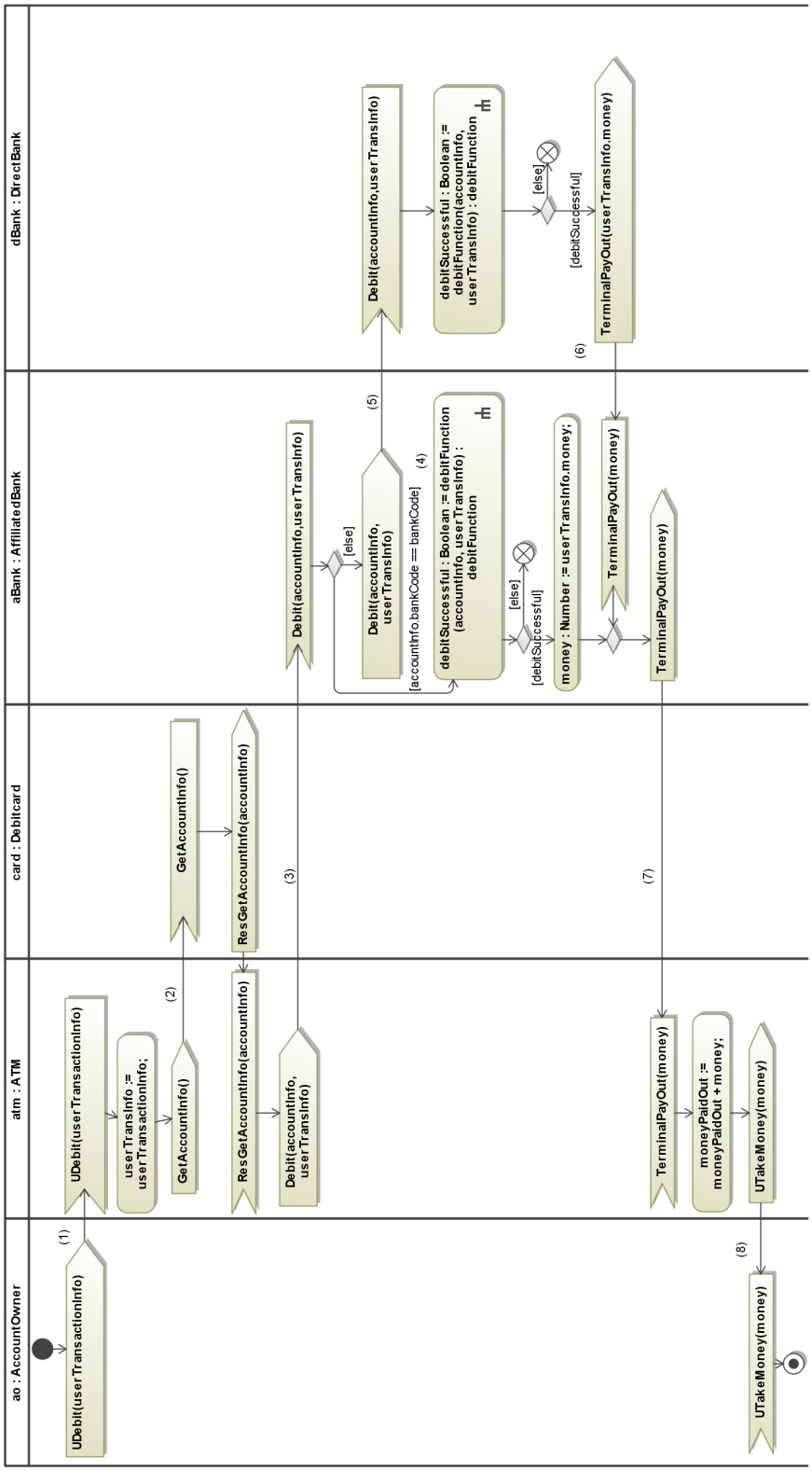


Figure 4: Protocol to withdraw money from an ATM

account belongs to the bank itself it will call the *debitFunction* (4) and then depending on the return value the money will be issued or not. But if it does not belong to this bank but to the *DirectBank*, the message will be forwarded to the *DirectBank* (5) and the amount will be debited there. After that, if the debit action was successful, a message will be sent back (6)(7) and the *ATM* will pay out the money (8).

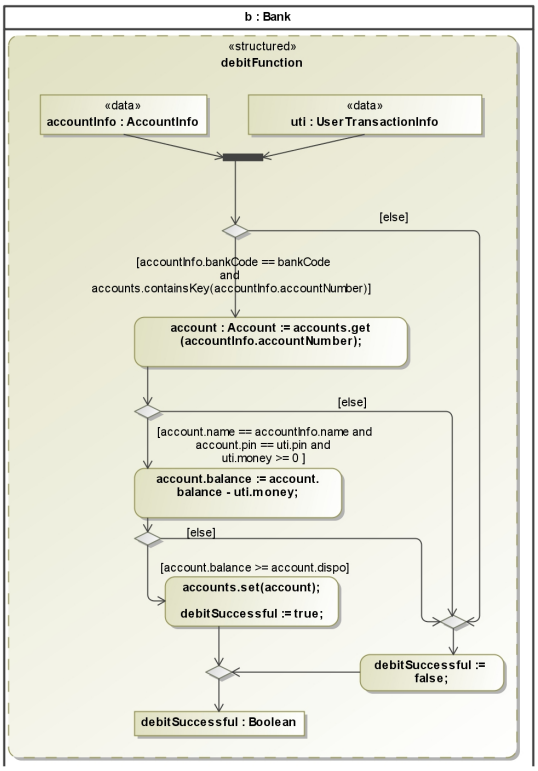


Figure 5: Sub-Activity debitFunction

In Figure 5 the sub-activity *debitFunction* is modelled. This functionality is defined for the upper class *Bank* and can be used from all sub classes. The bank receives the account information *accountInfo* (name, account number, bank code) and the user transaction information *uti* (PIN, money). Then it checks if the received bank code is equal to its own and if an account with the received account number exists. If the check was positive, a copy from the account will be took from the list of accounts and checked if the name and the PIN from the account are equal with the received data and if *money* is not negative. If this check is also positive the account balance will be decreased by *money* and the new account balance is checked to be not smaller than the *dispo* value. After the check, the account is updated by setting the account to the accounts list and true is returned. If one of the checks is negative, the return value will be false.

Another functionality supported by our example application Debitcard is the online transaction. Before the online transaction can be invoked, a handshake has to be processed.

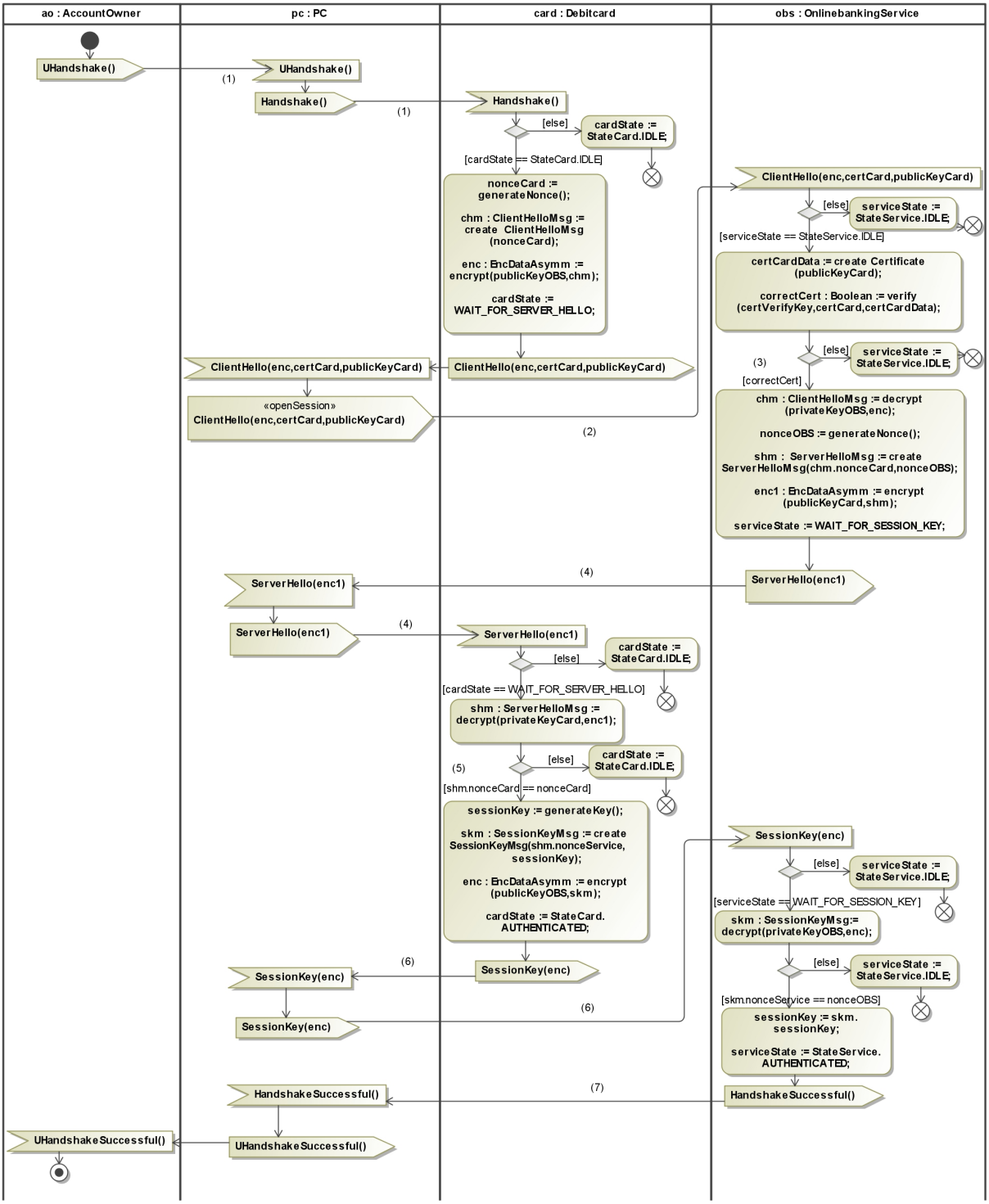


Figure 6: Handshake Protocol

In Figure 6 the handshake protocol is shown. It establishes a secure session between *Debitcard* and *OnlinebankingService*. This protocol uses security data types defined in SecureMDD to secure the messages. We use private and public keys as well as nonces and states to exchange a session key to secure messages over an intermediate like in this example the *PC* component. There are four participants involved in this scenario. The *AccountOwner ao*, the *PC pc*, the *Debicard card* and the *OnlinebankingService obs*. First the message *UHandshake* is sent over the *pc* to *card* (1). Then if the state is *IDLE*, a fresh nonce will be created, wrapped in the class *ClientHelloMsg* and encrypted with the public key *publicKeyOBS*. Then the state will be changed and the encrypted nonce together with a certificate, consisting of the signature of the *card's* public key and the *card's* public key self, will be sent over *pc* to *obs* (2). But before *pc* can forward this message, it is necessary to know which instance of *OnlinebankingService* should be used. Because *obs* has to store session dependent information like states and the exchanged session key it is not possible to use the same instance for all invokers. For that *OnlinebankingService* is stateful and provides a new instance for every invoker. Because the same instance should also be used in another protocols, it is necessary to model the first call of a stateful service. Therefore the stereotype ≪openSession≫ is supported and can be applied to a send signal action. Now *card* communicates with its own *OnlinebankingService* instance *obs*. After *obs* receives the message, it also checks if its state is *IDLE* and verifies the received certificate. If the the certificate is valid (3), *obs* decrypts the received nonce, creates another one, wraps the two nonces in the class *ServerHelloMsg* and encrypts it with the received public key. After that, the state will be changed and the encrypted nonces are sent to *card* (4). *card* checks the state and ensures that the last received message was *Handshake* and that no exception occurred. It encrypts the received nonces and checks that the first nonce is equal to the nonce that was generated in the previous card step (5). If the comparison was successful, the *card* creates a session key, stores it, wraps the session key and the nonce created by *obs* in *SessionKeyMsg* and encrypts it with the public key that belongs to *obs*. After that the card state is set to *AUTHENTICATED* and the session key together with the nonce will be sent to *obs* (6). *obs* receives the message, checks its state, encrypts the message, compares the nonce, stores the session key, sets its state to *AUTHENTICATED* and returns a non-security critical message *HandshakeSuccessful* (7). The used messages in this protocol are depicted in Figure 11 and Figure 9.

Figure 7 shows the protocol to transfer money from one account to another one. After the session key is exchanged and the states are set to *AUTHENTICATED*, the online transaction can be processed. The account owner (card holder) has to type in the transaction data, namely the PIN, the amount of money that should be transferred and the receiver account information. This data *(uoti)* will be sent over *pc* to the *card* (1). The *card* checks that the card state is *AUTHENTICATED*, wraps *uoti* and the account information (*accountInfo*) that is initially stored on the card in the message *TransactionData* and encrypts it with the previously exchanged session key. Then the state is set to *IDLE* and the encrypted data is sent to *obs* over the pc (2). After *obs* receives the message *TransactionData* it checks that its state is also *AUTHENTICATED* and decrypts the received data with the previously exchanged session key. The state is set back to *IDLE* to avoid replay attacks and on the basis of the card holder's bank code it is checked which bank the card holder account

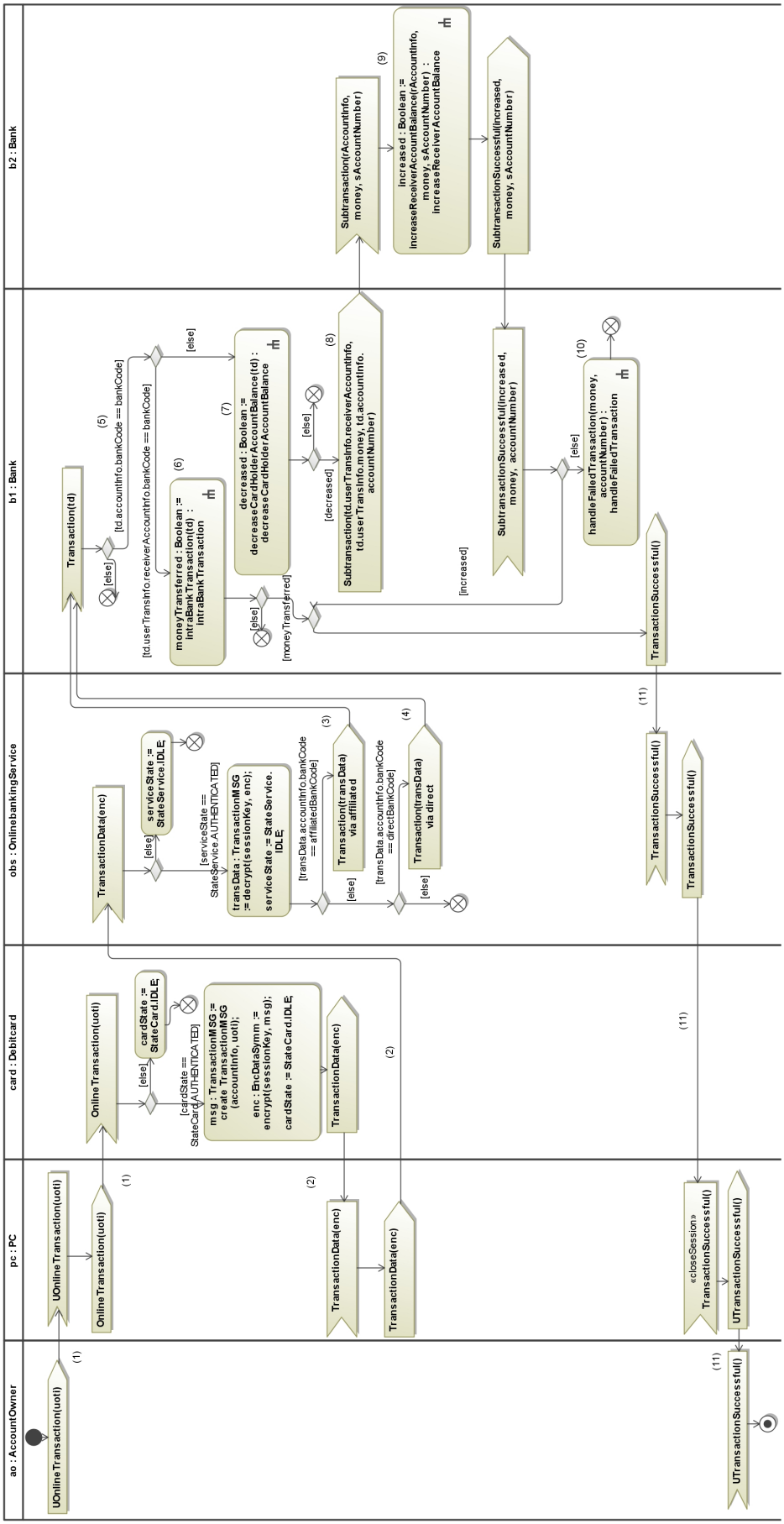


Figure 7: Protocol to transfer money from one account to another

belongs to. Dependent on this the message is sent over the port affiliated to the affiliated bank (3) or over the port directly to the direct bank (4). The ports are modelled in the deployment diagram (see Figure 2). If bank *b1* receives a transaction message with PIN, the amount of money that should be transferred, the sender account information and the receiver account information it checks if the card holder account belongs to it (5). If so, it checks if the receiver account also belongs to it and processes the transaction internally (6). Otherwise, the bank deducts the sum from the card holder account (7) and sends a request to the bank of the receiver account (8) to increase the sum in its account (9). If this fails (e.g. because of a non-existent account number) bank *b1* is notified, and the deduction from the card holder account is revoked (10). The failure (denoted by the flow final) will be propagated back to the user. Otherwise, the transaction was successful and a message is sent back over *obs* and pc to the user (11). But before the notification is sent to the user, pc closes the session with *obs* using the stereotype ≪closeSession≫.

An online transaction between different banks must avoid that money is deducted from one account but not deposited to another account. For this it is necessary that the messages during a transaction always arrive in the correct order. This is guaranteed in our example because the attacker has only the ability to read messages on the connection between affiliated bank and direct bank (see Figure 2, with a full Dolev-Yao attacker we cannot ensure that a protocol ever finishes). Because this connection is secured with Transport Layer Security (TLS) [6] the attacker can only read the encrypted messages. The used messages in this protocol are shown in Figure 12 and Figure 9.

In Figure 7 we have seen the use of the superclass *Bank* in an activity diagram. This is very useful to avoid the modelling of redundant behavior and makes the diagrams clearer. The deployment diagram (see Figure 2) ensures that if *b1* is a *DirectBank* then *b2* is an *AffiliatedBank* and vice versa. Same functionality is encapsulated in methods that are predefined or designed in the model. Methods designed in the model like *intraBankTransaction* (see Figure 8), *decreaseCardHolderAccountBalance* (see Figure 13), *increaseReceiverAccountBalance* (see Figure 14) and *handleFailedTransaction* (see Figure 15) allow big and complex protocols to be divided into smaller diagrams, so that all of them remain clear.

The modelled method *intraBankTransaction* (see Figure 8) represents a part of our online transaction protocol that handles the transaction where the card holder account and the receiver account belong to the same bank. First we cache some information from the object we have received through the input parameter and then we check whether the accounts are available in the store *accounts*. Then we get the accounts and check if the PIN of that card holder account is equal to the PIN entered by the user, whether the amount is positive and make sure that the card holder does not try to transfer more money than he is allowed to. If it is true, the money will be subtracted from the card holder account and added to the receiver account. Otherwise nothing will happen and the method returns false.

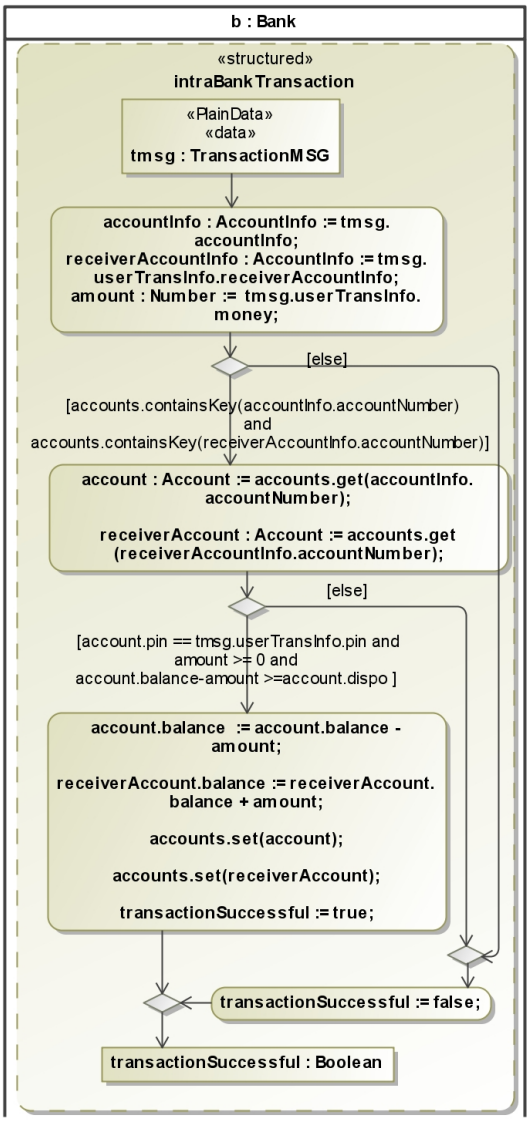


Figure 8: Sub-Activity IntraBankTransaction

# 5 Executable Code

An advantage of SecureMDD compared to some of the other model-driven approaches (see Section 7) is that runnable code will be generated. To achieve that, our approach is to model the whole system. That is possible because we use our own platform independent language (MEL) to describe the behaviour of the components.

The service components in SecureMDD are implemented as Web Services that use SOAP [25] as underlying technology. To implement Web Services we use Metro[3] that integrates JAX-WS[23] (Java API for XML - Web Services). JAX-WS is a standard and supports server and clients as well as the Code-First principle with plain old Java objects (POJO) which is very important because we already generate the code. Services will be invoked by stubs that are automatically generated by the *wsimport* library, that is a part of JAX-WS. These stubs manage the communication between client and service, the mapping between Java objects and XML documents as well as the transmitting and receiving of XML documents.

To support services that provide a new instance for every invoker, stateful services that are a part of JAX-WS are used. To communicate with a stateful service instance, it is necessary to invoke a stateless service that gives the invoker the address of a stateful service instance. That means every invoker gets a fresh instance for a session but it does not protect against an attacker who could invoke that service instance, too.

```
@WebService
public class AffiliatedBank {

  private String bankCode;
  private ListOfAccount accounts;
  ...

  public AffiliatedBank()
      throws java.lang.Exception { ... }

  public AffiliatedBank(ListOfAccount accounts, String bankCode)
      throws java.lang.Exception { ... }

  @WebMethod
  public MessageWrapper process(MessageWrapper msg)
      throws ServiceException { ... }

  private Message processMessage(Message inmsg)
      throws java.lang.Exception { ... }

  private Message sendMsg(Message msg, int port,
            Boolean openSessionBeforeSend,
            Boolean closeSessionAfterReceive)
                throws java.lang.Exception { ... }

  public void startService() throws java.lang.Exception { ... }

  public void stopService() { ... }

  ...

}
```

Listing 1: Generated code for AffiliatedBank

Listing 1 depicts the generated class for *AffiliatedBank* from the Debitcard example. The class is annotated as service with *@WebService* and the methods

[3] http://metro.java.net/

that can be called from outside and describe the interface are annotated with *@WebMethod*. The class attributes contain the attributes from the class *AffiliatedBank* in the class diagram (see Figure 3) and some technical ones. An empty and non-empty constructor is generated. The empty one is for JAXB (JAX-Binding) that is responsible for mapping between code and XML documents and the non-empty one gets the initialize attributes. The method *process* receive incoming messages and invokes *processMessage* with the message that is wrapped in the *MessageWrapper* object. The wrapping is necessary because JAXB needs a container to transmit an upper class with the information about its subclass. The *processMessage* is shown in Listing 2.

```
private Message processMessage(Message inmsg) throws java.lang.Exception {
  switch (inmsg.getCode()) {
    case Code.DEBIT :
      return processDebit((Debit) inmsg);
    case Code.TERMINALPAYOUT :
      return processTerminalPayOut((TerminalPayOut) inmsg);
    ...
    default :
      stop();
      return null;
  }
}
```

Listing 2: processMessage from AffilatedBank

Dependent on the message code that describes the subclass, the appropriate process method is called. For example, for the message *Debit* the method *processDebit* is called. This method is generated from the swimlane *AffiliatedBank* in the activity diagram in Figure 4 and is depicted in Listing 3.

```
  private Message processDebit(Debit inmsg) throws java.lang.Exception {
    synchronized (this) {
      AccountInfo accountInfo = inmsg.getAccountInfo();
      UserTransactionInfo userTransInfo = inmsg.getUserTransInfo();
      if (accountInfo.getBankCode().equals(bankCode)) {
        boolean debitSuccessful = debitFunction(accountInfo,
            userTransInfo);
        if (debitSuccessful) {
          int money = userTransInfo.getMoney();
          return sendMsg(new TerminalPayOut(money));
        } else {
          stop();
          return null;
        }
      } else {
        return sendMsg(new Debit(accountInfo, userTransInfo),
            Ports.AffiliatedBank2DirectBank_default);
      }
    }
  }
```

Listing 3: processDebit from AffilatedBank

The generated code is similar to the modelled behaviour and is generated automatically. To send a message the method *sendMsg* is called. In Listing 4 the generated send method of *AffiliatedBank* is shown. This method is called by every send of a message and encapsulates the different behaviour to send a message because we support smart card and service communication and need a central place to manipulate the send messages.

```
private Message sendMsg(Message msg, int port) throws java.lang.Exception {
  Message response;
  switch (port) {
    case Ports.AffiliatedBank2DirectBank_default :
```

```
        DirectBank directBank = new DirectBankService()
            .getDirectBankPort();
        setTimeout((BindingProvider) directBank);
        response = directBank.process(new MessageWrapper(msg)).getMsg();
        return processMessage(response);
      default :
        stop();
        return null;
    }
}
```

Listing 4: Generated code for invoking a service

The *AffiliatedBank* can only communicate with a *DirectBank*. That information is modelled in the deployment diagram (see Figure 2) and the port *AffiliatedBank2DirectBank_default* is generated automatically and describes to which component a message should be sent. A timeout will be set to avoid that the invoker trying to connect or send a message infinitely. To invoke a service stubs are used. Hence, the whole communication between a client and a service can be accessed through an attribute like in this example *directBank*. But the stubs that are generated by *wsimport*, contain only the class attributes and method signatures but not the body of the methods. Therefore, only the part that is responsible for the communication is used. The other part that contains the classes is deleted and replaced with the classes that are generated by SecureMDD.

Because a service can be concurrently invoked by many clients, parallelism is an important issue. If a write access to an attribute depends on previously read attribute, an optimal synchronization is a non trivial task. To manage the parallelism problem, our first solution is to synchronize the areas between receiving and sending a message (see Listing 3). The synchronization of a stateful service is done by a static variable and synchronizes the areas for all instances of that service. For stateless services it is sufficient to synchronize on the service instance self. Because this solution implements the behaviour of our formal model (see Section 6), dead locks and live locks that influence the security properties will be identified during the verification. Through this synchronization, the parallelism problems are solved but any kind of concurrent invocation is disabled.

To support the functionality to get an object by an identifier we provide the functions *set*, *get* and *containsKey* on a key-value storage. For example the accounts in the Debitcard case study are realized with this prototypically implemented data structure. To deploy that banking system (Debitcard) the store *accounts* will be probably realized with a database. Also, the terminals should be accessible with a graphical user interface (GUI). For that, the key-value storage implementation is encapsulated in a class that can be replaced by another implementation and for a GUI we provide a user interface which is used to notify a GUI when a message is received.

Because during software development requirements can change, it is useful that code for the modelled applications can be automatically generated and tested. For that, a test case which initializes all instances, deploys all services and calls the user messages to execute the protocols has to be created; to deploy the services we generate the methods *startService* and *stopService* for each service; this way we can call *startService* before and *stopService* after a test case. In our test framework the services will be deployed on a light-weight server that is integrated in Java but of course we can also deploy the services on other servers.

To test a service application we have to generate stubs for all components that call services. Hence after all component code is generated, we deploy the services and generate stubs for every client (either a service or a terminal) on its local platform. The whole process can be invoked with one click inside Eclipse.

A modelled application consists of different components that will be deployed in different environments. This influences the service communication because the generated stubs depend on the service address. In our test environment all services are deployed on the localhost. If a service is deployed on another host with a different IP or domain, the stubs of the invoker need to be changed. To minimize effort and because we want to generate runnable code which must not be edited, we encapsulate the functionality of stubs generation for each component in a class which allows setting the IP/domain-address.

TLS uses keys and certificates, thus we need a key and trust store to provide them. The key store contains asymmetric key pairs while the trust store provides signed certificates; to use TLS, keys and certificates have to be transferred inside a secure environment before the system can be deployed.

For Debitcard, the full generated and runnable code can be found on our website[4].

# 6 Formal Verification

An important aspect of the SecureMDD approach is the support for the formal verification of security properties of a system under development. This is achieved by the automatic generation of a formal model that is suitable for our interactive theorem prover KIV [13, 1, 10]. The formal model is based on algebraic specifications and Abstract State Machines (ASMs) [3]. It specifies a world in which agents exchange messages according to the protocols, and an attacker tries to break the security. Agents are either users, smart cards, terminals, or services, and there exists an arbitrary (finite) number of agents. In this world protocol runs between arbitrary agents take place that may happen in parallel or consecutively. The idea is that the formal model models the real world where many users use ATMs or perform online banking at the same time. One step of the ASM corresponds to one protocol step (receiving and processing a message), a run of the ASM creates a trace, i.e. a sequence of steps, that describes one possibility of what can happen in this world. Since the ASM is indeterministic it models not one but many traces. The idea is that the ASM models everything that can happen in the real world (with respect to the application). So if the application can be proved secure in the formal model it should be secure in the real world. More details can be found in [19]. In the following we will describe only how services are modelled formally.

## 6.1 Transport Layer Security and the Attacker

In Section 4 we have shown that security stereotypes for connections like *TransportLayerSecurity* influence the stereotype *Threat*. If an attacker has the abilities to *read*, *send*, and *suppress* messages (i.e. a Dolev-Yao attacker [7] for this connection) and the connection is secured with TLS, the attacker loses some of those abilities. Because TLS is a secure protocol we use some of its security

[4]http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/

properties [6]. Messages are encrypted with a session key, their integrity is ensured by a message authentication code (MAC), and a sequence number is used to detect missing or replayed messages. If an error is detected the connection is closed.

This means the attacker can only read messages that are encrypted with a prior exchanged session key that will never be used in another session. Therefore reading the messages is useless for the attacker because they cannot be used for replays (as described below) and we are only concerned with logical security properties, not traffic analysis where the message length or timing may be important.

Furthermore, in the formal model the attacker loses his ability to send messages, because if the message is not encrypted with the correct session key (which the attacker does not possess) the MAC verification will fail and the message will not be accepted. A replayed message is encrypted with the correct session key, but will not be accepted because of the sequence number. The ability to suppress messages is lost as well because the next message will have an incorrect sequence number. However, the attacker has the ability to terminate the connection as described above.

To summarize, it is appropriate to formalize a TLS secured connection as one where an attacker can either do nothing or can only abort the connection (depending on the annotations in the deployment diagram Figure 2).

## 6.2 Stateful and Stateless Services as Agents

A service component can be stateless or stateful. Both must be treated slightly different in the formal model. A stateless service is similar to other agents like terminals and smart cards. The formal model may have an arbitrary number of services or it may be restricted to exactly one.

A stateful service however creates an instance of itself for each invoker. The actual code of the invoker calls a manager which is also implemented as a stateless service. It creates an instance of the actual service, deploys it and returns the address of this service instance. This behavior is not modelled in the UML model of the application, and it is not reflected in the formal specification because it is an implementation issue only and does not change the modelled behavior or the security aspects of the modelled application. Thus, only the address of the service is transmitted which is not a security-critical information leak because the attacker already knows all components in the formal model.

For one stateful service the formal model has an arbitrary number of agents that represent the different new instances of the same service. They all have the same initial attributes that are reset with every connection establishment. Thus a stateful service is modeled as a set of agents that can be handled as the other agent types.

## 6.3 Verification of Security Properties

Beside the automatic code generation, the verification of security properties for the generated applications is a major benefit for the development of secure systems. With this approach we do not have to guess whether the application is secure, since we were able to formally verify it.

Usually only generic properties like secrecy or authentication are proven for security protocols (see [22] for an overview). In contrast, the SecureMDD approach focuses on application specific security properties [20]. They give better confidence in the properties of the application as a whole. In the bank application from Section 4 it is interesting to know that PINs and session keys remain secret, but the real properties are about money. For example, the bank application has the property that the amount of money remains constant in the following sense:

> The sum of all account balances plus the amount of all the money that has been withdrawn from cash machines is unchanged in all runs of the Abstract State Machine (ASM).

The proof works by proving an invariant for every step of the ASM. It turns out the property stated above is not quite correct because not yet finished protocol runs must be taken into account. For example, during an online transaction (see Figure 7) there is a situation where one account has been debited, but the receiver not yet credited. In a sense, the money is contained in the message between the different banks, and must be included in the money count. So the actual property written as a Hoare triple (and slightly simplified) is:

money = ATMMoney(atms)
+ DirectBankMoney(accounts)
+ AffiliatedBankMoney(accounts)
+ MoneyInTransit(messages) →
{ ASM(messages, atms, accounts, ...) }
money = ATMMoney(atms)
+ DirectBankMoney(accounts)
+ AffiliatedBankMoney(accounts)
+ MoneyInTransit(messages)

This property is proved formally with the KIV system. The main difficulty is to specify MoneyInTransit correctly. If during a proof attempt it turns out that the specification is not correct, it must be modified, and the proofs must done again. This can happen even if the protocol is secure.

To verify that security property it was also necessary to consider different kinds of connections. The connection between terminal and smartcard is inherently unreliable because a user can whip the card out of the reader at any time. In a more general setting, other types of connections should be supported. In the debitcard application, the connections between *ATM* and *AffiliatedBank* as well as between AffiliatedBank and DirectBank should not be disconnected before the protocol is finished. Otherwise money can be lost and the mentioned security property does not hold. For that we specified the critical connections as not disconnect-able.

# 7 Related Work

There are some approaches related to our SecureMDD that differ in certain aspects. For example UMLSec developed by Jan Jürjens [12] as well as SecureUML by David Basin et. al. [2] are both approaches to develop security-critical systems with extended UML by using an UML profile. SecureUML is tailored to

role-based access control applications and additionally supports specific authorization constraints with OCL. UMLSec allows to express security properties like secrecy, integrity and role-based access control with stereotypes. Both consider only standard security properties that can be proved with a model checker or automated theorem prover whereas we are able to prove application specific properties. Furthermore, both do not generate executable code. In contrast the SecureUML approach generates access control infrastructures for Enterprise JavaBeans.

MDD4SOA developed by Mayer et. al. [15] is a model-driven approach for service orchestration that transforms a platform independent model into several platform specific models and those to code for the languages BPEL, WSDL, Java and the formal language Jolie. It uses its own UML profile (UML4SOA [14]) to provide the modelling of service-oriented architectures and also verify properties with the formal language Jolie. But it does not consider security-critical protocols and does not define a language like our MEL to describe actions. They also do not generate executable code.

Another approach developed by Deubler et al. [5] considers the development of security-critical service-oriented systems. For modelling and verification it uses the tool AUTOFOCUS [11] that provides an own modelling language similar to UML. The considered security mechanisms are authentication and authorization that are proved with a model checker.

We are not aware of an approach like ours that provides model-driven development for security-critical systems that consider services, terminals and smart cards, generates executable code and verifies application specific security properties for a modelled system.

# 8 Conclusion and Future Work

Many security-critical systems consist of components that interact with each other over a network to perform tasks together. A common standard to implement this are Web Services. In this paper we have presented an enhancement of our SecureMDD approach to consider secure service application. Now we are able to model secure service and smart card applications and automatically generate runnable code as well as prove application specific security properties. With services the communication structure is enhanced greatly. Every kind of component has to be handled in a special way. Services are realized as SOAP Web Services and can be stateless or stateful. Connections to services can be secured with a predefined stereotype ≪TLS≫ as well as with application specific protocols. With a banking system named Debitcard we demonstrated the abilities of our SecureMDD approach.

The next steps are to support WS-Security to include concepts like Kerberos or SAML and use a standard that is readable by other services that are not created by our approach. We also will handle parallelism in a more efficient way, integrate a real database and extend our communication structure. Furthermore we will support OCL expressions to define security properties within the platform independent model for the formal verification and integrate an existing model checker.

# References


[1] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

[2] D. Basin, J. Doser, and T. Lodderstedt. Model Driven Security: From UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology*, pages 39–91, 2006.

[3] E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[4] Z. Chen. *Java card technology for smart cards: architecture and programmer's guide*. Prentice Hall PTR, 2000.

[5] M. Deubler, J. Grünbauer, J. Jürjens, and G. Wimmel. Sound development of secure service-based systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 115–124. ACM, 2004.

[6] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. IETF Network Working Group, August 2008. URL: `http://www.ietf.org/rfc/rfc5246.txt`.

[7] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. In *Proc. 22th IEEE Symposium on Foundations of Computer Science*. IEEE, 1981.

[8] Gartner. Gartner says worldwide enterprise software revenue to grow 9.5 percent in 2011, 2011. Available online at `http://www.gartner.com/it/page.jsp?id=1728615`.

[9] T. S. Group. Chaos report, 2009. Available online at `http://www.standishgroup.com/newsroom/chaos_2009.php`.

[10] D. Haneberg, S. Bäumler, M. Balser, H. Grandy, F. Ortmeier, W. Reif, G. Schellhorn, J. Schmitt, and K. Stenzel. The User Interface of the KIV Verification System — A System Description. *Electronic Notes in Theoretical Computer Science UITP special issue*, 2006.

[11] F. Huber, S. Molterer, A. Rausch, B. Schatz, M. Sihling, and O. Slotosch. Tool supported specification and simulation of distributed systems. In *Software Engineering for Parallel and Distributed Systems, 1998. Proceedings. International Symposium on*, pages 155–164. IEEE, 1998.

[12] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[13] KIV homepage. `http://www.informatik.uni-augsburg.de/swt/kiv`.

[14] N. Koch, P. Mayer, R. Heckel, L. G
"onczy, and C. Montangero. D1. 4a: UML for service-oriented systems. *Specification, SENSORIA Project*, 16004, 2007.

[15] P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-Driven Service Orchestration. In *Proceedings of 12th IEEE International EDOC Conference (EDOC 2008)*, September 15-19, 2008, Munich, Germany.

[16] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. Model-Driven Code Generation for Secure Smart Card Applications. In *20th Australian Software Engineering Conference*. IEEE Press, 2009.

[17] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications. In *2009 International Conference on Availability, Reliability and Security*, pages 841–846. IEEE, 2009.

[18] N. Moebius, K. Stenzel, and W. Reif. Modeling Security-Critical Applications with UML in the SecureMDD Approach. *International Journal On Advances in Software*, 1(1), 2008.

[19] N. Moebius, K. Stenzel, and W. Reif. Generating Formal Specifications for Security-Critical Applications - A Model-Driven Approach . In *ICSE 2009 Workshop: International Workshop on Software Engineering for Secure Systems (SESS'09)*. IEEE/ACM Digital Libary, 2009.

[20] N. Moebius, K. Stenzel, and W. Reif. Formal verification of application-specific security properties in a model-driven approach. In *Proceedings of ESSoS 2010 - International Symposium on Engineering Secure Software and Systems*. Springer LNCS 5965, 2010.

[21] O. M. G. (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*, 2011.

[22] J. C. L. Pimental and R. Monroy. Formal support to security protocol development: A survey. *Computacion y Sistemas*, 12(1), 2008.

[23] R. M. Roberto Chinnici, Marc Hadley. *The Java API for XML-Based Web Services (JAX-WS) 2.0*. JCP, 2006. Available online at `http://jcp.org/aboutJava/communityprocess/final/jsr224/index.html`.

[24] Sun Microsystems Inc. *Java Card 2.2 Specification*, 2002. `http://java.sun.com/products/javacard/`.

[25] W3C. *SOAP Version 1.2*, 2007. Available online at `http://www.w3.org/TR/soap12-part0/`.

# A Diagrams for Debitcard

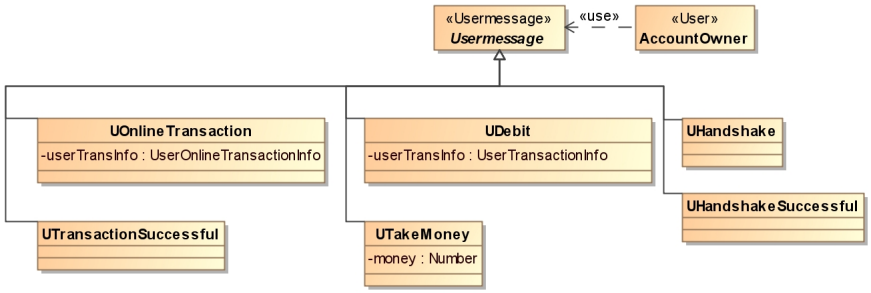


Figure 9: User Messages

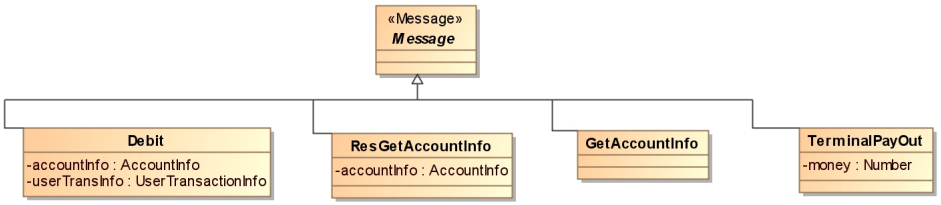


Figure 10: Messages for withdrawal of money from an ATM

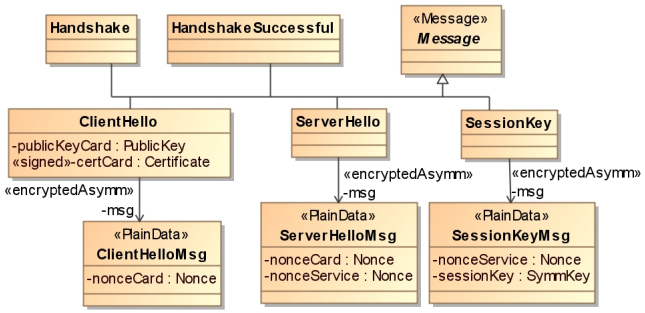


Figure 11: Messages for the handshake protocol

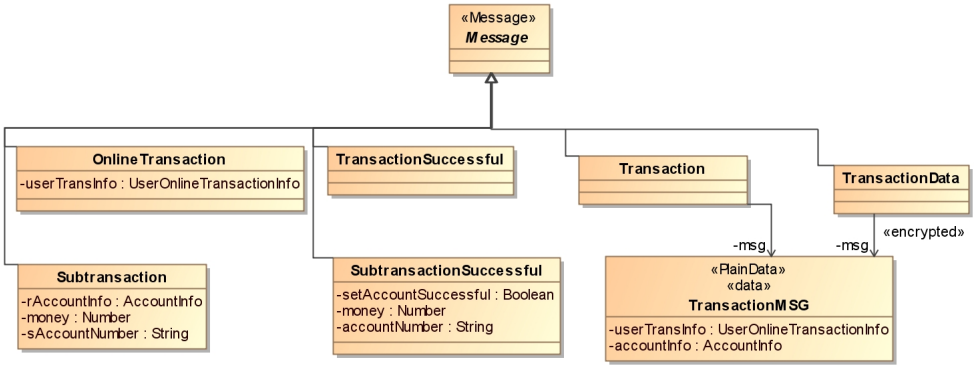


Figure 12: Messages for online transaction

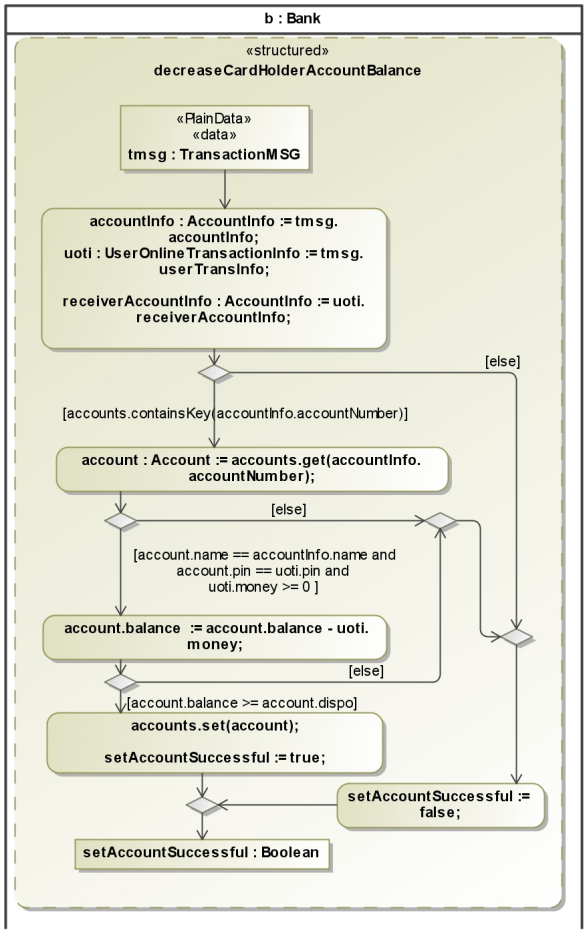


Figure 13: Sub-Activity to decrease card holder account balance

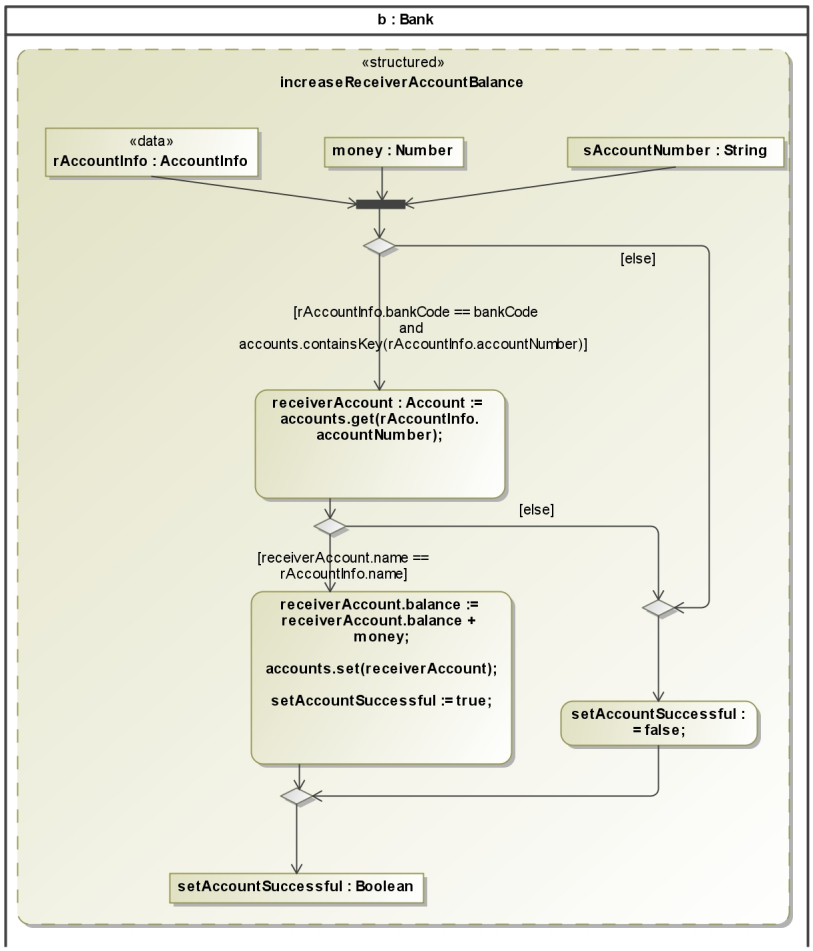


Figure 14: Sub-Activity to increase card holder account balance

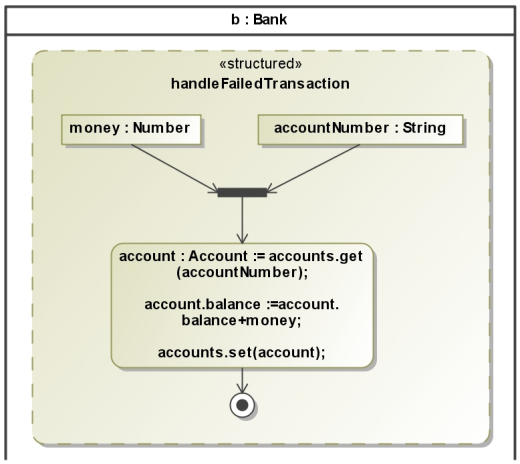


Figure 15: Sub-Activity to handle a failed transaction